INTRODUCTION TO PYTHON & PYTHON 101

AIOTLAB, September 2023

Table of Contents

1. Reference
2. Why Python?
3. Installation
4. Introduction to Python
   - Data Types
   - Python Variables
   - Multiple Variables
   - Output Variables
   - Variable Names
   - Numbers
   - Strings, String formatting
   - Booleans
   - Casting
   - For loops
   - If...Else and While loops
   - Function

Table of Contents

1. Reference
2. Why Python?
3. Installation
4. Introduction to Python
   - Data Types
   - Python Variables
   - Multiple Variables
   - Output Variables
   - Variable Names
   - Numbers
   - Strings, String formatting
   - Booleans
   - Casting
   - For loops
   - If...Else and While loops
   - Function

Reference

- W3School Python Tutorial: https://www.w3schools.com/python/default.asp
- Python for Data Analysis, Wes McKinney: https://wesmckinney.com/book/

Table of Contents

1. Reference
2. Why Python?
3. Installation
4. Introduction to Python
   - Data Types
   - Python Variables
   - Multiple Variables
   - Output Variables
   - Variable Names
   - Numbers
   - Strings, String formatting
   - Booleans
   - Casting
   - For loops
   - If...Else and While loops
   - Function

Why Python?

- Python has been one of the most popular interpreted programming languages, along with Perl, Ruby, and others.
- Python is also known as a scripting language which can be used to quickly write small programs, or scripts to automate other tasks.
- Python has developed a large and active scientific computing and data analysis community.
- Python has gone from a bleeding-edge or ”at your own risk” scientific computing language to one of the most important languages for data science, machine learning, and general software development in academia and industry.
- Has many libraries and framework suitable for analyzing data and working with machine learning, deep learning such as scikit-learn, pandas, NumPy, TensorFlow, PyTorch.

Table of Contents

1. Reference
2. Why Python?
3. Installation
4. Introduction to Python
   - Data Types
   - Python Variables
   - Multiple Variables
   - Output Variables
   - Variable Names
   - Numbers
   - Strings, String formatting
   - Booleans
   - Casting
   - For loops
   - If...Else and While loops
   - Function

Installation

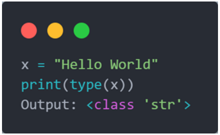
- Download Python at https://www.python.org/downloads/
- Download Anaconda and follow installation guideline at https://conda.io/projects/conda/en/stable/user-guide/install/download.html

Table of Contents

1. Reference
2. Why Python?
3. Installation
4. Introduction to Python
    - Data Types
    - Python Variables
    - Multiple Variables
    - Output Variables
    - Variable Names
    - Numbers
    - Strings, String formatting
    - Booleans
    - Casting
    - For loops
    - If...Else and While loops
    - Function

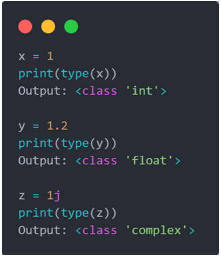
Data Types

- Text type: str (string)



```
x = "Hello World"
print(type(x))
Output: <class 'str'>
```

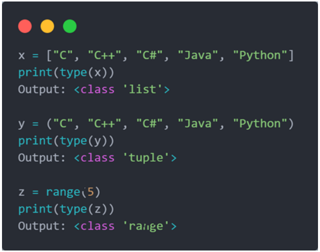
- Numeric types: int (integer), float, complex



```
x = 1
print(type(x))
Output: <class 'int'>

y = 1.2
print(type(y))
Output: <class 'float'>

z = 1j
print(type(z))
Output: <class 'complex'>
```

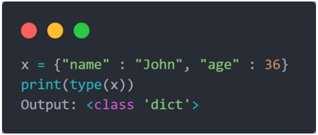
Data Types

- Sequence types: list, tuple, range



```
x = ["C", "C++", "C#", "Java", "Python"]
print(type(x))
Output: <class 'list'>

y = ("C", "C++", "C#", "Java", "Python")
print(type(y))
Output: <class 'tuple'>

z = range(5)
print(type(z))
Output: <class 'range'>
```

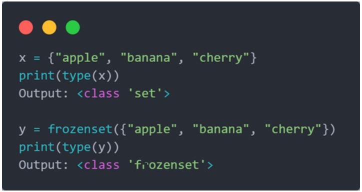
- Mapping type: dict



```
x = {"name" : "John", "age" : 36}
print(type(x))
Output: <class 'dict'>
```

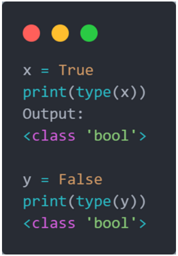
Data Types

- Set types: set, frozenset



```
x = {"apple", "banana", "cherry"}
print(type(x))
Output: <class 'set'>

y = frozenset({"apple", "banana", "cherry"})
print(type(y))
Output: <class 'frozenset'>
```

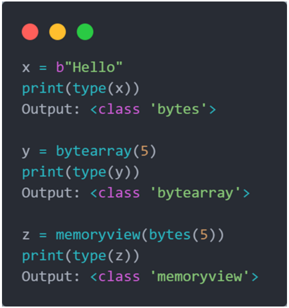
- Boolean type: bool (boolean)



```
x = True
print(type(x))
Output:
<class 'bool'>

y = False
print(type(y))
<class 'bool'>
```

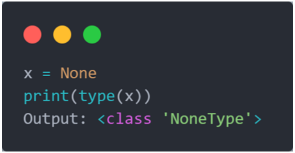
Data Types

- Binary types: bytes, bytearray, memoryview



```
x = b"Hello"
print(type(x))
Output: <class 'bytes'>

y = bytearray(5)
print(type(y))
Output: <class 'bytearray'>

z = memoryview(bytes(5))
print(type(z))
Output: <class 'memoryview'>
```

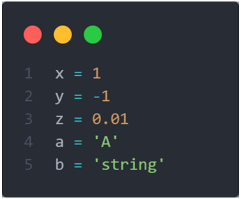
- None type: NoneType



```
x = None
print(type(x))
Output: <class 'NoneType'>
```

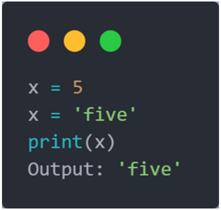
Python Variables

- Python has no command for declaring a variable.
- A variable is created the moment you first assign a value to it.



```
x = 1
y = -1
z = 0.01
a = 'A'
b = 'string'
```

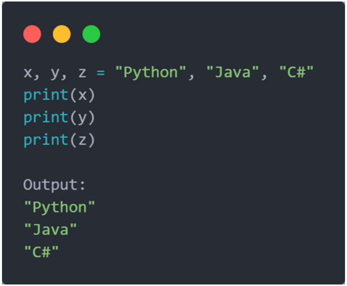
- Variables do not need to be declared with any particular type and can even change type after they have been set.



```
x = 5
x = 'five'
print(x)
Output: 'five'
```

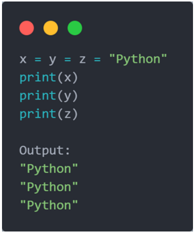
Multiple Variables

- Many values to multiple variables



```
x, y, z = "Python", "Java", "C#"
print(x)
print(y)
print(z)

Output:
"Python"
"Java"
"C#"
```

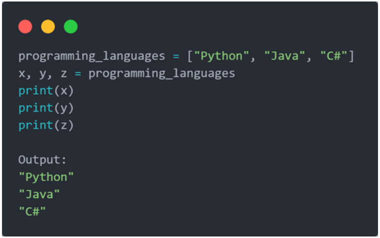
- One value to multiple variables



```
x = y = z = "Python"
print(x)
print(y)
print(z)

Output:
"Python"
"Python"
"Python"
```

Multiple Variables

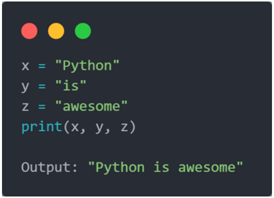
- Unpack a Collection: If you have a collection of values in a list, tuple etc. Python allows you to extract the values into variables. This is called unpacking.



```
programming_languages = ["Python", "Java", "C#"]
x, y, z = programming_languages
print(x)
print(y)
print(z)

Output:
"Python"
"Java"
"C#"
```

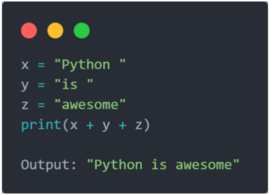
Output Variables

- The Python **print()** function is often used to output variables.
- In the **print()** function, you output multiple variables, separated by a comma:



```
x = "Python"
y = "is"
z = "awesome"
print(x, y, z)

Output: "Python is awesome"
```

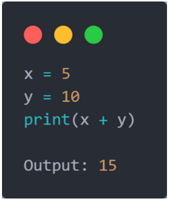
- You can also use the + operator to output multiple variables:



```
x = "Python "
y = "is "
z = "awesome"
print(x + y + z)

Output: "Python is awesome"
```

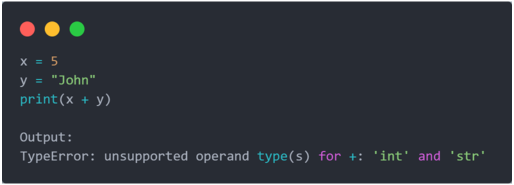
Output Variables

- For numbers, the + character works as a mathematical operator:



```
x = 5
y = 10
print(x + y)

Output: 15
```

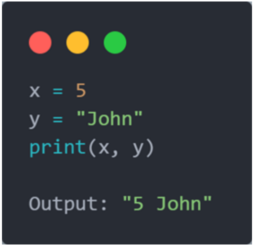
- In the **print()** function, when you try to combine a string and a number with the + operator, Python will give you an error:



```
x = 5
y = "John"
print(x + y)

Output:
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Output Variables

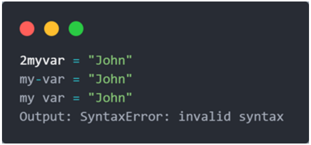
- The best way to output multiple variables in the **print()** function is to separate them with commas, which even support different data types:



```
x = 5
y = "John"
print(x, y)

Output: "5 John"
```

Variable Names

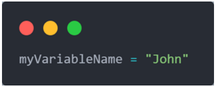
- Rules for Python variables:
  - A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume).
  - A variable name must start with a letter or the underscore character.
  - A variable name cannot start with a number.
  - A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).
  - Variable names are case-sensitive (age, Age and AGE are three different variables).
  - A variable name cannot be any of the Python keywords.
- Example of illegal variable names:



```
2myvar = "John"
my-var = "John"
my var = "John"
Output: SyntaxError: invalid syntax
```

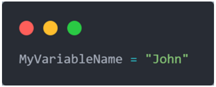
Variable Names

- Multi Words Variable Names:
    - Camel Case



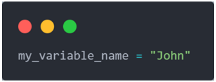
```
myVariableName = "John"
```

    - Pascal Case



```
MyVariableName = "John"
```

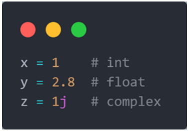
    - Snake Case



```
my_variable_name = "John"
```

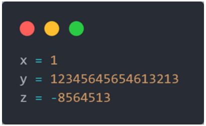
Numbers

- There are three numeric types in Python: int, float, and complex.



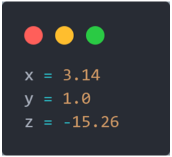
```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
```

- Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.



```
x = 1
y = 12345645654613213
z = -8564513
```

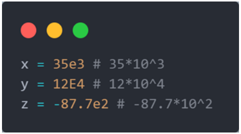
- Float, or ”floating point number” is a number, positive or negative, containing one or more decimals.



```
x = 3.14
y = 1.0
z = -15.26
```

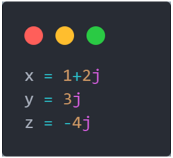
Numbers

- Float can also be scientific numbers with an ”e” to indicate the power of 10.



```
x = 35e3 # 35*10^3
y = 12E4 # 12*10^4
z = -87.7e2 # -87.7*10^2
```

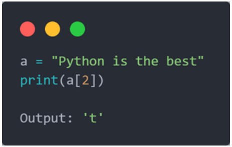
- Complex numbers are written with a ”j” as the imaginary part.



```
x = 1+2j
y = 3j
z = -4j
```

Strings, String formatting

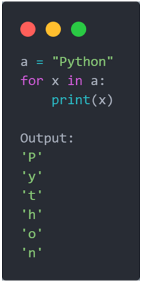
- Strings in python are surrounded by either single quotation marks, or double quotation marks. E.g., ‘Python’ is the same as “Python”.
- Strings are Arrays:
  - Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.
  - However, Python does not have a character data type, a single character is simply a string with a length of 1.
  - Square brackets can be used to access elements of the string.



```
a = "Python is the best"
print(a[2])

Output: 't'
```

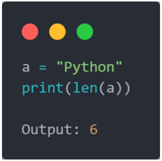
Strings, String formatting

- Since strings are arrays, we can loop through the characters in a string, with a for loop.



```
a = "Python"
for x in a:
    print(x)

Output:
'P'
'y'
't'
'h'
'o'
'n'
```

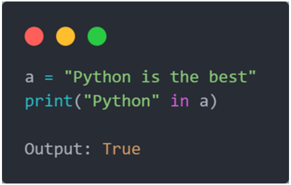
- To get the length of a string, use the len() function.



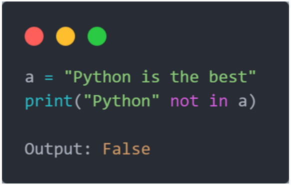
```
a = "Python"
print(len(a))

Output: 6
```

Strings, String formatting

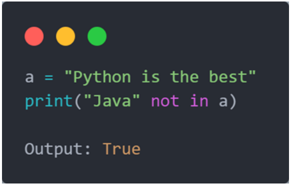
- To check if a certain phrase or character is present in a string, we can use the keyword **in**.



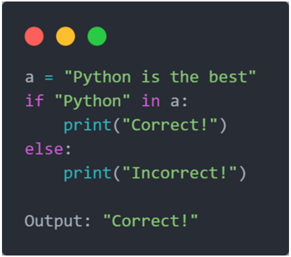
```
a = "Python is the best"
print("Python" in a)

Output: True
```



```
a = "Python is the best"
print("Python" not in a)

Output: False
```



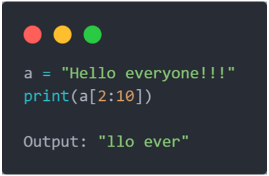
```
a = "Python is the best"
print("Java" not in a)

Output: True
```



```
a = "Python is the best"
if "Python" in a:
    print("Correct!")
else:
    print("Incorrect!")

Output: "Correct!"
```

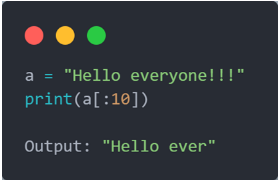
Strings, String formatting

- Slicing Strings
  - You can return a range of characters by using the slice syntax.
  - Specify the start index and the end index, separated by a colon, to return a part of the string.



```
a = "Hello everyone!!!"
print(a[2:10])

Output: "llo ever"
```

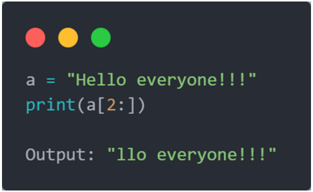
  - Slice from the Start



```
a = "Hello everyone!!!"
print(a[:10])

Output: "Hello ever"
```

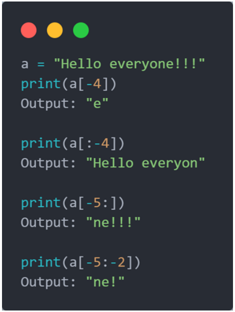
Strings, String formatting

- Slice to the End



```
a = "Hello everyone!!!"
print(a[2:])

Output: "llo everyone!!!"
```

- Negative Indexing: Use negative indexes to start the slice from the end of the string.



```
a = "Hello everyone!!!"
print(a[-4])
Output: "e"

print(a[:-4])
Output: "Hello everyon"

print(a[-5:])
Output: "ne!!!"

print(a[-5:-2])
Output: "ne!"
```

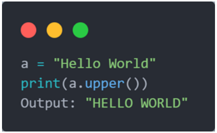
Strings, String formatting

Question

What is the result of “a[-2:-5]”?

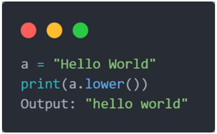
Strings, String formatting

- Modify Strings: Python has a set of built-in methods that can be used on strings.
  - Upper Case



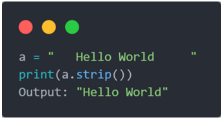
```
a = "Hello World"
print(a.upper())
Output: "HELLO WORLD"
```

  - Lower Case



```
a = "Hello World"
print(a.lower())
Output: "hello world"
```

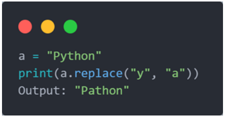
  - Remove Whitespace before and after the actual text.



```
a = "   Hello World     "
print(a.strip())
Output: "Hello World"
```

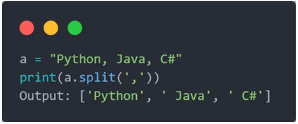
Strings, String formatting

- Replace strings



```
a = "Python"
print(a.replace("y", "a"))
Output: "Pathon"
```

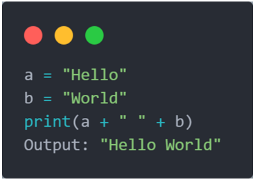
- Split strings: Return a list where the text between the specified separator becomes the list items



```
a = "Python, Java, C#"
print(a.split(','))
Output: ['Python', ' Java', ' C#']
```

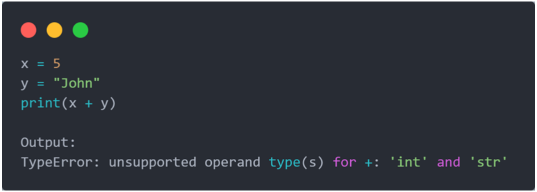
Strings, String formatting

- Concatenate Strings: Strings can be combined by using + operator between them.



```
a = "Hello"
b = "World"
print(a + " " + b)
Output: "Hello World"
```

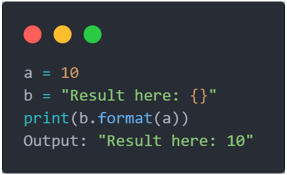
  - As stated earlier, strings and numbers cannot be combined by directly using + operator



```
x = 5
y = "John"
print(x + y)

Output:
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

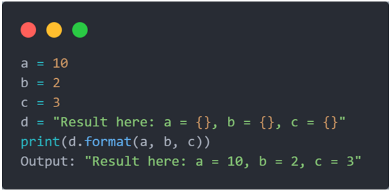
Strings, String formatting

- But we can combine strings and numbers by using the **format()** method! The **format()** method takes the passed arguments, formats them, and places them in the string where the placeholders {} are



```
a = 10
b = "Result here: {}"
print(b.format(a))
Output: "Result here: 10"
```

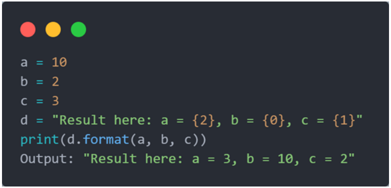
- The **format()** method takes unlimited number of arguments, and are placed into the respective placeholders.



```
a = 10
b = 2
c = 3
d = "Result here: a = {}, b = {}, c = {}"
print(d.format(a, b, c))
Output: "Result here: a = 10, b = 2, c = 3"
```

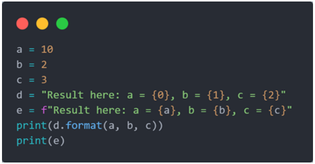
Strings, String formatting

- You can add index inside the brackets to specify the order of arguments to be placed in the string.



```
a = 10
b = 2
c = 3
d = "Result here: a = {2}, b = {0}, c = {1}"
print(d.format(a, b, c))
Output: "Result here: a = 3, b = 10, c = 2"
```

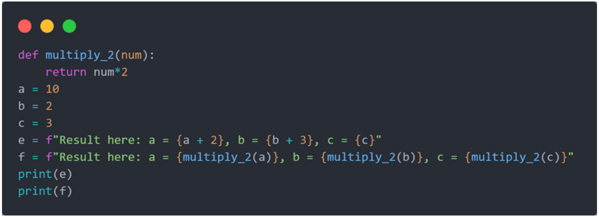
- F-Strings: A new way to format strings



```
a = 10
b = 2
c = 3
d = "Result here: a = {0}, b = {1}, c = {2}"
e = f"Result here: a = {a}, b = {b}, c = {c}"
print(d.format(a, b, c))
print(e)
```

Strings, String formatting

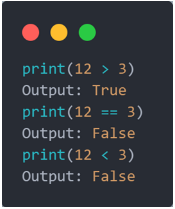
- You can also pass an operation or a function returning value inside the curly braces of f-strings in case your results need to be preprocessed first.



```python
def multiply_2(num):
    return num*2
a = 10
b = 2
c = 3
e = f"Result here: a = {a + 2}, b = {b + 3}, c = {c}"
f = f"Result here: a = {multiply_2(a)}, b = {multiply_2(b)}, c = {multiply_2(c)}"
print(e)
print(f)
```

Booleans

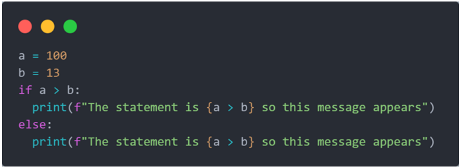
- Booleans represent one of two values: **True** or **False**.
- In programming you often need to know if an expression is **True** or **False**.
- You can evaluate any expression in Python, and get one of two answers, **True** or **False**.
- When you compare two values, the expression is evaluated and Python returns the Boolean answer.



```
print(12 > 3)
Output: True
print(12 == 3)
Output: False
print(12 < 3)
Output: False
```

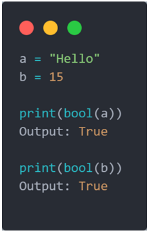
Booleans

- When you run a condition in an if statement, Python returns **True** or **False**.



```
a = 100
b = 13
if a > b:
  print(f"The statement is {a > b} so this message appears")
else:
  print(f"The statement is {a > b} so this message appears")
```

- Evaluate Values and Variables: The **bool()** function allows you to evaluate any value, and give you **True** or **False** in return.



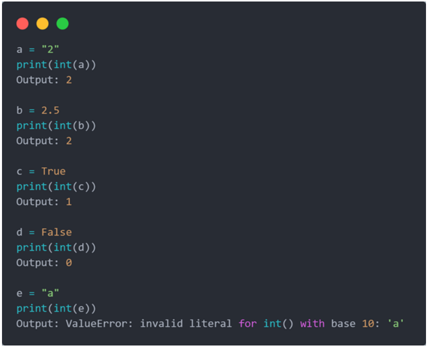
```
a = "Hello"
b = 15

print(bool(a))
Output: True

print(bool(b))
Output: True
```

Booleans

- Most values are **True** except for some values such as: False, 0, None, empty strings, empty lists, empty arrays, empty dicts, empty tuples.

Casting

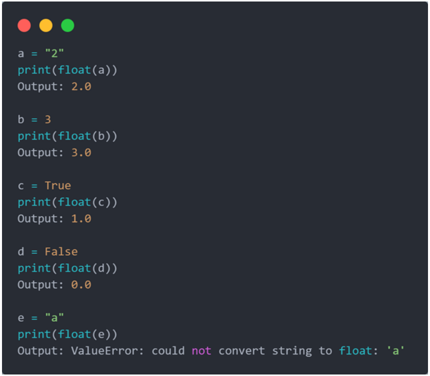
- There may be times when you want to specify a type on to a variable. This can be done with casting.
- Casting in python can be done by using functions:
  - **int()**: constructs an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number).



```
a = "2"
print(int(a))
Output: 2

b = 2.5
print(int(b))
Output: 2

c = True
print(int(c))
Output: 1

d = False
print(int(d))
Output: 0

e = "a"
print(int(e))
Output: ValueError: invalid literal for int() with base 10: 'a'
```

Casting

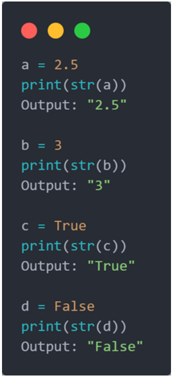
- **float()**: constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer).



```
a = "2"
print(float(a))
Output: 2.0

b = 3
print(float(b))
Output: 3.0

c = True
print(float(c))
Output: 1.0

d = False
print(float(d))
Output: 0.0

e = "a"
print(float(e))
Output: ValueError: could not convert string to float: 'a'
```

Casting

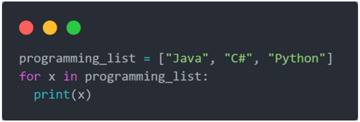
- **str()**: constructs a string from a wide variety of data types, including strings, integer literals and float literals.



```
a = 2.5
print(str(a))
Output: "2.5"

b = 3
print(str(b))
Output: "3"

c = True
print(str(c))
Output: "True"

d = False
print(str(d))
Output: "False"
```

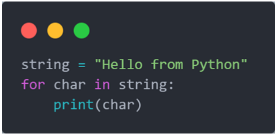
For loops

- A **for** loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
- The **for** loop in Python does not require an indexing variable to set beforehand.
- Looping through a list



```
programming_list = ["Java", "C#", "Python"]
for x in programming_list:
  print(x)
```

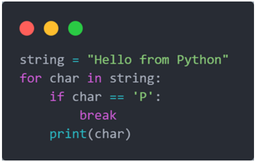
- Looping through a string



```
string = "Hello from Python"
for char in string:
    print(char)
```

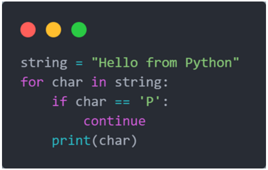
For loops

- Using **break** statement during a loop will stop the loop immediately.



```python
string = "Hello from Python"
for char in string:
    if char == 'P':
        break
    print(char)
```

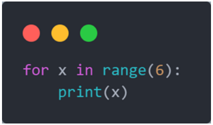
- Using **continue** statement during a loop will bypass the current element/index and move to the next element/index.



```python
string = "Hello from Python"
for char in string:
    if char == 'P':
        continue
    print(char)
```

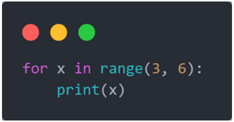
For loops

- The **range()** function:
  - To loop through a set of code a specified number of times, we can use the **range()** function.
  - The **range()** function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.



```
for x in range(6):
    print(x)
```

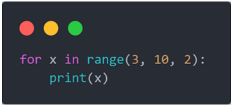
  - The **range()** function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter.



```
for x in range(3, 6):
    print(x)
```

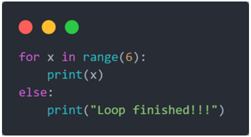
For loops

- The **range()** function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter.



```python
for x in range(3, 10, 2):
    print(x)
```

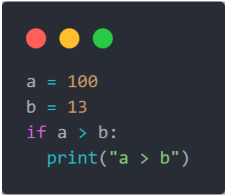
- **Else** in **For** loop: The else keyword in a for loop specifies a block of code to be executed when the loop is finished.



```python
for x in range(6):
    print(x)
else:
    print("Loop finished!!!")
```

If...Else

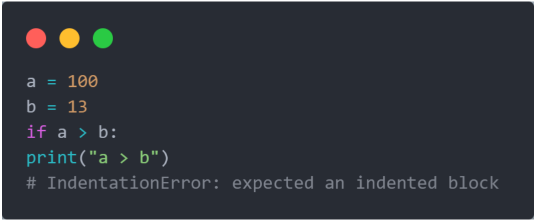
- Python supports the usual logical conditions from mathematics:
  - Equals: **a == b**
  - Not Equals: **a != b**
  - Less than: **a < b**
  - Less than or equal to: **a <= b**
  - Greater than: **a > b**
  - Greater than or equal to: **a >= b**
- These conditions can be used in several ways, most commonly in "if statements" and loops.
- An "if statement" is written by using the **if** keyword.



```
a = 100
b = 13
if a > b:
  print("a > b")
```

If...Else

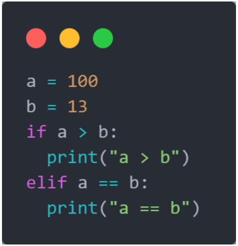
- Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.



```
a = 100
b = 13
if a > b:
print("a > b")
# IndentationError: expected an indented block
```

If...Else

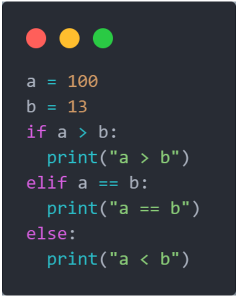
- The **elif** keyword is Python’s way of saying ”if the previous conditions were not true, then try this condition”.



```
a = 100
b = 13
if a > b:
  print("a > b")
elif a == b:
  print("a == b")
```

If...Else

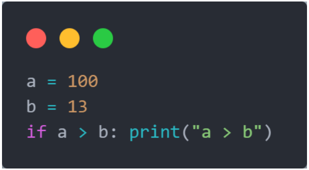
- The **else** keyword catches anything which isn't caught by the preceding conditions.



```python
a = 100
b = 13
if a > b:
  print("a > b")
elif a == b:
  print("a == b")
else:
  print("a < b")
```

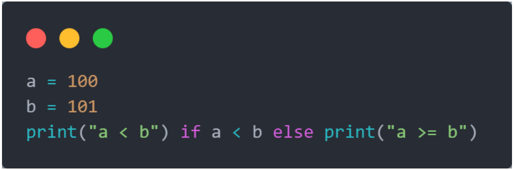
If...Else

- Short Hand If: If you have only one statement to execute, you can put it on the same line as the if statement.



```python
a = 100
b = 13
if a > b: print("a > b")
```

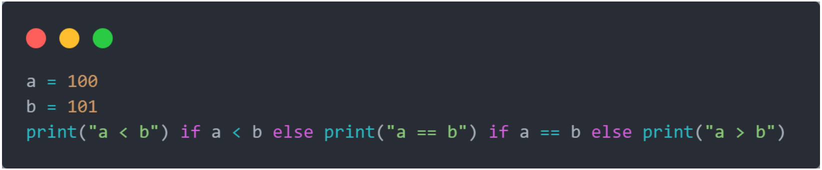
- Short Hand If ... Else (Ternary Operators or Conditional Expressions): If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:



```python
a = 100
b = 101
print("a < b") if a < b else print("a >= b")
```

If...Else

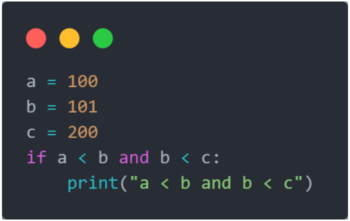
You can also have multiple else statements on the same line:



```
a = 100
b = 101
print("a < b") if a < b else print("a == b") if a == b else print("a > b")
```

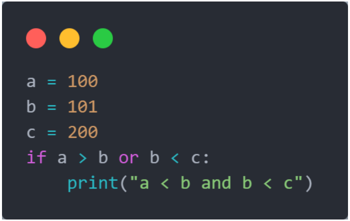
If...Else

- The **and** keyword is a logical operator, and is used to combine conditional statements:



```python
a = 100
b = 101
c = 200
if a < b and b < c:
    print("a < b and b < c")
```

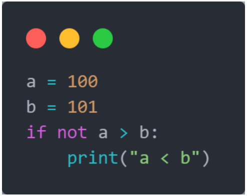
- The **or** keyword is a logical operator, and is used to combine conditional statements:



```python
a = 100
b = 101
c = 200
if a > b or b < c:
    print("a < b and b < c")
```

If...Else

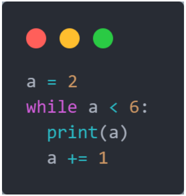
- The **not** keyword is a logical operator, and is used to reverse the result of the conditional statement:



```
a = 100
b = 101
if not a > b:
    print("a < b")
```

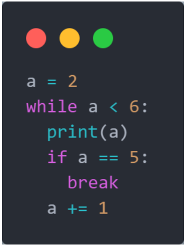
While Loops

- With the **while** loop we can execute a set of statements as long as a condition is true.



```python
a = 2
while a < 6:
  print(a)
  a += 1
```

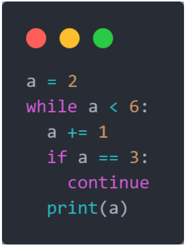
- With the **break** statement we can stop the loop even if the while condition is true:



```python
a = 2
while a < 6:
  print(a)
  if a == 5:
    break
  a += 1
```

While Loops

- With the **continue** statement we can stop the current iteration, and continue with the next:

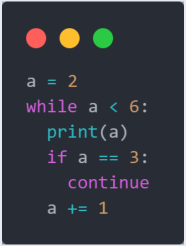


```
a = 2
while a < 6:
  a += 1
  if a == 3:
    continue
  print(a)
```

While Loops

Question

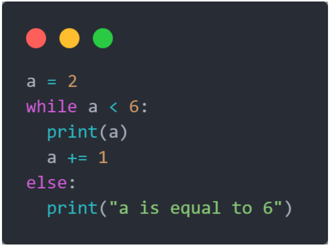
What is the result of the below code block?



```python
a = 2
while a < 6:
  print(a)
  if a == 3:
    continue
  a += 1
```

While Loops

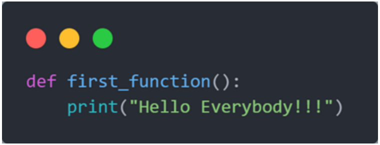
- With the **else** statement we can run a block of code once when the condition no longer is true:



```
a = 2
while a < 6:
  print(a)
  a += 1
else:
  print("a is equal to 6")
```

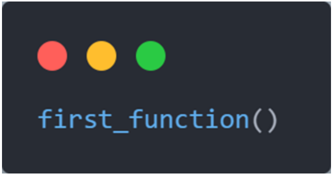
Function

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.
- Creating a function: function is defined using the **def** keyword.



```python
def first_function():
    print("Hello Everybody!!!")
```

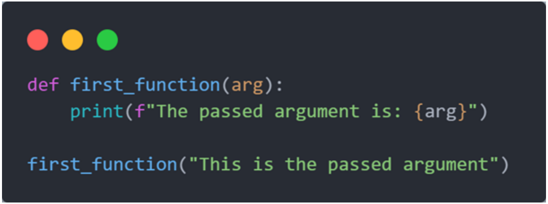
- You can call a function by using its name followed by parenthesis.



```python
first_function()
```

Function

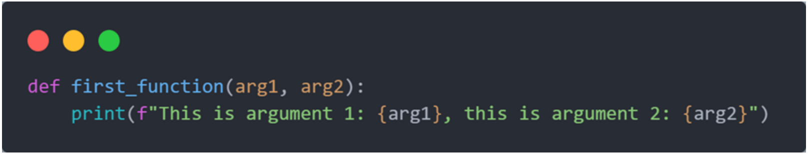
- Arguments:
  - Can be called as parameters.
  - Information can be passed into functions as arguments.
  - Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.



```python
def first_function(arg):
    print(f"The passed argument is: {arg}")

first_function("This is the passed argument")
```

Function

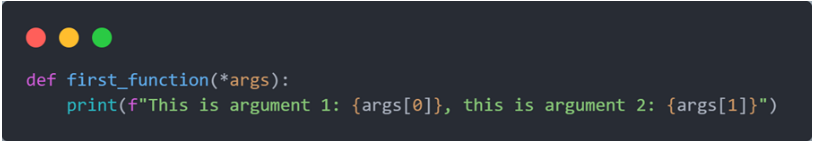
- By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.



```
def first_function(arg1, arg2):
    print(f"This is argument 1: {arg1}, this is argument 2: {arg2}")
```

Function

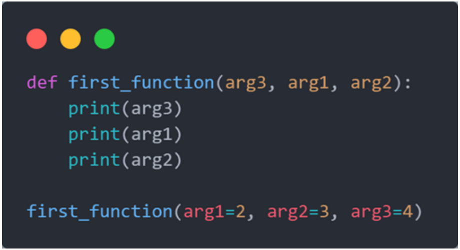
- If you do not know how many arguments that will be passed into your function, add a ***** before the parameter name in the function definition to make it become an Arbitrary Argument (***args**).
    - This way the function will receive a tuple of arguments, and can access the items accordingly.



```
def first_function(*args):
    print(f"This is argument 1: {args[0]}, this is argument 2: {args[1]}")
```

Function

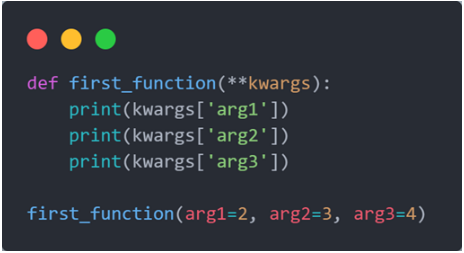
- Keyword Arguments (**kwargs**): The arguments can be passed to function with the format of "key = value". This way the order of the arguments does not matter.



```
def first_function(arg3, arg1, arg2):
    print(arg3)
    print(arg1)
    print(arg2)

first_function(arg1=2, arg2=3, arg3=4)
```

Function

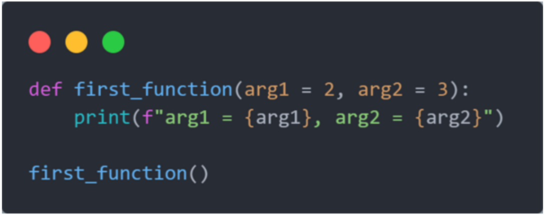
- Arbitrary Keyword Arguments (****kwargs**): If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ****** before the parameter name in the function definition. This way the function will receive a dictionary of arguments, and can access the items accordingly.



```python
def first_function(**kwargs):
    print(kwargs['arg1'])
    print(kwargs['arg2'])
    print(kwargs['arg3'])

first_function(arg1=2, arg2=3, arg3=4)
```

Function

- Default Parameter Value: when defining a function the arguments can have their own default values by passing the value to them inside the parenthesis at the **def** line of the function. By doing this we don't have to worry if the parameter have default value has been passed to the function or not.



```python
def first_function(arg1 = 2, arg2 = 3):
    print(f"arg1 = {arg1}, arg2 = {arg2}")

first_function()
```

The End

THE END!!!